Evaluation of LH*LH for a Multicomputer Architecture

Andy D. Pimentel and Louis O. Hertzberger

Dept. of Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Abstract. Scalable, distributed data structures can provide fast access to large volumes of data. In this paper, we present a simulation study in which the performance behaviour of one of these data structures, called LH*LH, is evaluated for a multicomputer architecture. Our experimental results demonstrate that the access-time to LH*LH can be very small and does not deteriorate for increasing data structure sizes. Furthermore, we also show that parallel access to the LH*LH data structure may speed up client applications. This speed-up is, however, shown to be constrained by a number of effects.

1 Introduction

Modern database applications require fast access to large volumes of data. Sometimes the amount of data is so large than it cannot be efficiently stored or processed by a uniprocessor system. Therefore, a *distributed data structure* can be used that distributes the data over a number of processors within a parallel or distributed system. This is an attractive possibility because the achievements in the field of communication networks for parallel and distributed systems have made remote memory accesses faster than accesses to the local disk [3]. So, even when disregarding the additional processing power of parallel platforms, it has become more efficient to use the main memory of other processors than to use the local disk.

It is highly desirable for a distributed data structure to be scalable. The data structure should not have a theoretical upper limit after which performance degrades (i.e. the time to access data is independent of the number of stored data elements) and it should grow and shrink incrementally rather than reorganising itself totally on a regular basis (e.g. rehashing of all distributed records). For distributed memory multicomputers, a number of Scalable Distributed Data Structures (*SDDS*s) have been proposed [7]. In these distributed storage methods, the processors are divided into *clients* and *servers*. A client manipulates the data by inserting data elements, searching for them or removing them. A server stores a part of the data, called a *bucket*, and receives data-access requests from clients. Generally, there are three ground rules for the implementation of an SDDS in order to realize a high degree of scalability [4]:

- The SDDS should not be addressed using a central directory which forms a bottleneck.
- Each client should have an image of how data is distributed which is as accurate as possible. This image should be improved each time a client makes an "addressing error", i.e. contacts a server which does not contain the required data. The client's state (its current image) is only needed for efficiently locating the remote data; it is not required for the correctness of the SDDS's functionality.

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 217–228, 1999.

– If a client has made an addressing error, then the SDDS is responsible for forwarding the client's request to the correct server and for updating the client's image.

For an efficient SDDS, it is essential that the communication needed for data operations (retrieval, insertion, etc.) is minimised while the amount of data residing at the server nodes (i.e. the *load factor*) is well balanced. In [7], Litwin et al. propose an SDDS, called *LH**, which addresses the issue of low communication overhead and balanced server utilisation. This SDDS is a generalisation of Linear Hashing (LH) [6], which will be elaborated upon in the next section. For LH*, insertions usually require one message (from client to server) and three messages in the worst case. Data retrieval requires one extra message as the requested data has to be returned.

In this paper, we evaluate the performance of a variant of the LH* SDDS, called LH*LH, which was proposed by Karlsson [4, 5]. For this purpose, we use a simulation model which is based on the architecture of a Parsytec CC multicomputer. With this model, we investigate how scalable the LH*LH SDDS actually is and which factors affect its performance. The reason for our interest in the LH*LH SDDS (and the Parsytec CC) is that LH*LH is considered for using in a parallel version of the Monet database [1] which will initially be targeted towards the Parsytec CC multicomputer.

The next section explains the concept of Linear Hashing. In Section 3, we discuss the distributed data structure LH*LH. Section 4 describes the simulation model we have used in this study. In Section 5, our experimental results are presented. Finally, Section 6 concludes the paper.

2 Linear Hashing

Linear Hashing (LH) [6] is a method to dynamically manage a table of data. More specifically, it allows the table to grow or shrink in time without suffering from a penalty with respect to the space utilisation or the access time. The LH table is formed by $N \times 2^i + n$ buckets, where N is the number of starting buckets ($N \geq 1$ and $n < 2^i$). The meaning of i and n is explained later on. The buckets in the table are addressed by means of a pair of hashing functions h_i and h_{i+1}, with $i = 0, 1, 2...$ Each bucket can contain a predefined number of data elements. The function h_i hashes data keys to one of the first $N \times 2^i$ buckets in the table. The function h_{i+1} is used to hash data keys to the remaining buckets. In this paper, we assume that the hash functions are of the form

$$h_i(\text{key}) \ \rightarrow \ \text{key } \textit{mod } (N \times 2^i) \tag{1}$$

The LH data structure grows by splitting a bucket into two buckets whenever there is a *collision* in one of the buckets, which means that a certain load threshold is exceeded. Which bucket has to be split is determined by a special pointer, referred to as n. The actual splitting involves three steps: creating a new bucket, dividing the data elements over the old and the newly created bucket and updating the pointer n. Dividing the data elements over the two buckets is done by applying the function h_{i+1} to each element in the splitting bucket. The n pointer is updated by applying $n = (n + 1) \ mod \ N \times 2^i$. Indexing the LH data structure is performed using both h_i and h_{i+1}. Or, formally:

$$\begin{gathered} \text{index}_{bucket} = h_i(\text{key}) \\ \text{if } (\text{index}_{bucket} < n) \text{ then } \text{index}_{bucket} = h_{i+1}(\text{key}) \end{gathered} \tag{2}$$

As the buckets below the n pointer have been split, these buckets should be indexed using h_{i+1} rather than with h_i. When the n pointer wraps around (because of the modulo), i should be incremented.

The process of shrinking is similar to the growing of the LH data structure. Instead of splitting buckets, two buckets are merged whenever the load factor drops below a certain threshold. In this study, we limit our discussion to the splitting within SDDSs.

3 The LH*LH SDDS

The LH* SDDS is a generalisation of Linear Hashing (LH) to a distributed memory parallel system [7]. In this study, we focus on one particular implementation variant of LH*, called LH*LH [4, 5]. The LH*LH data is stored over a number of server processes and can be accessed through dedicated client processes. These clients form the interface between the application and LH*LH. We assume that each server stores *one* LH*LH bucket of data, which implies that a split always requires the addition of an extra server process. Globally, the servers apply the LH* scheme to manage their data, while the servers use traditional LH for their local bucket management. Thus, a server's LH*LH bucket is implemented as a collection of LH buckets. Hence, the name LH*LH. In Figure 1, the concept of LH*LH is illustrated.

As was explained in the previous section, addressing a bucket in LH is done using a key and the two variables i and n (see Equation 2). In LH*LH, the clients address the servers in the same manner. To do so, each client has its own *image* of the values i and n: i' and n' respectively. Because the images i' and n' may not be up to date, clients can address the wrong server. Therefore, the servers need to verify whether or not incoming client requests are correctly addressed, i.e. can be handled by the receiving server. If an incoming client request is incorrectly addressed, then the server forwards the request to the server that is believed to be correct. For this purpose, the server uses a forwarding algorithm [7] for which it is proven that a request is forwarded at most twice before the correct server is found. Each time a request is forwarded, the forwarding server

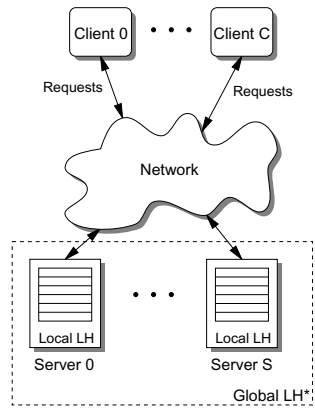


Fig. 1. The LH*LH SDDS.

sends a so-called Image Adjustment Message (IAM) to the requesting client. This IAM contains the server's local notion of i and n and is used to adjust the client's i' and n' in order to get them closer to the global i and n values. As a consequence, future requests will have a higher probability of being addressed correctly.

The splitting of an LH*LH bucket is similar to the splitting of LH buckets. The pointer n is implemented by a special token which is passed from server to server in the same manner as n is updated in LH: it is forwarded in a ring of the servers 0 to $N \times 2^i$, where N is the number of starting servers. When a server holds the n token and its load factor is larger than a particular threshold, the server splits its LH*LH bucket and forwards the n token. Splitting the LH*LH bucket is done by initialising a new server (by sending it a special message) and shipping half of its LH buckets to the new server (remember that the LH*LH bucket is implemented as a collection of LH buckets).

It has been shown in [7] that a splitting threshold which can be dynamically adjusted performs better than a static one. Therefore, LH*LH applies a dynamic threshold T which is based on an estimation of the global load factor [5]:

$$T = M \times V \times \frac{2^i + n}{2^i}$$

where M is a sensitivity parameter and V is the capacity of an LH*LH bucket in number of data elements. Typically, M is set to a value between 0.7 and 0.9. If the number of data elements residing at the server with the n token is larger than the threshold T, then the server splits its data.

4 The Simulation Model

The parallel architecture we focus on in this study is based on that of a Parsytec CC multicomputer. The configuration we have used for our initial experiments consists of 16 PowerPC 604 processors connected in a multistage network with an application-level throughput of 27 MByte/s for point-to-point communication. To model this multicomputer architecture, we have used the Mermaid simulation environment which provides a framework for the performance evaluation of multicomputer architectures [10, 11, 9].

In our model, a multicomputer node can contain either one server or client process. The clients provide the network part of the model with communication requests, which are messages to server processes containing commands that operate on the LH*LH SDDS (e.g. insert, lookup, etc.). In addition, the server processes can also issue communication requests, like when a client request has to be forwarded to another server.

The network model simulates a wormhole-routed network [2] at the flit-level and is configured to model a Mesh of Clos topology [10]. This is the generic topology from which the Parsytec CC's topology has been derived. The Mesh of Clos topology is a mesh network containing clusters of Clos multistage sub-networks. In the case of the Parsytec CC, a Mesh of Clos topology with a 1×1 mesh network is used, i.e. it contains a pure multistage network. For routing, the model uses deterministic XY-routing [8] for the mesh network and a deterministic scheme based on the source node's identity [10] for the multistage subnetworks. Furthermore, messages larger than 4Kb are split up in separate packets of 4Kb each.

We have validated our Parsytec CC network model against the real machine with a suite of small communication benchmarks and found that the error in predicted execution times for these programs is on the average less than 4% [9]. For *some* experiments, we were also able to compare our simulation results with the results from a real LH*LH implementation [4, 5]. Although the measurements for the real implementation were obtained using a multicomputer which is different from the one we have modelled, the comparison between our simulation results and the actual execution results can still give us some insight into the validity of our simulation model. We found that the simulation results correspond closely to the behaviour measured for the real implementation.

5 Experiments

We performed experiments with an application which builds the LH*LH SDDS using a Dutch dictionary of roughly 180,000 words as the data keys. The data elements that are being inserted consist of a key only (they do not have "a data body") unless stated otherwise. Throughout this paper, we use the term *blobsize* when referring to the size of the body of data elements. By default, our architecture model only accounts for the delays that are associated with communication. Computation performed by the clients and servers is not modelled. In other words, the clients and servers are infinitely fast. This should give us an upper bound of the performance of LH*LH when using the modelled network technology. However, we will also show several results of experiments in which we introduce and vary a computational server latency.

The first experiments concern LH*LH's performance on the 16-node Parsytec CC platform. Because the number of available nodes in this platform is rather limited, we needed to calibrate the split-threshold such that the LH*LH SDDS does not grow larger than the available number of nodes. Another result of the limited number of nodes is that the model only simulates up to 5 clients in order to reserve enough nodes for the server part of LH*LH. To overcome these problems, we have also simulated a larger multicomputer platform, of which the results are discussed later in this study. Throughout this paper, we assume that the number of starting servers is one, i.e. $N = 1$.

Figure 2a shows the predicted time it takes to build the LH*LH SDDS on the Parsytec CC when using c clients, where $c = 1, 2, ..., 5$. In the case multiple clients are used, they *concurrently* insert a different part of the dictionary. The data points in Figure 2a correspond to the points in time where LH*LH splits take place. The results show that the build-time scales linearly with the number of insertions. Thus, the insertion latency does not deteriorate for large data structure sizes (it is dominated entirely by the message round trip time). In this respect, Figure 2a clearly indicates that LH*LH is scalable.

The results of Figure 2a also show that the build-time decreases when more clients are used. This is due to the increased parallelism as each client concurrently operates on the LH*LH SDDS. In Figure 2b, the relative effect (i.e. the speedup) of the number of clients is shown. When increasing the number of clients, the obtained speedup scales quite well for the range of clients used.

Another observation that can be made is that the occurrence of splits (the data points in Figure 2a) is not uniformly distributed with respect to the insertions. Most of the splits are clustered in the first 20,000 insertions and near the 180,000 insertions. This

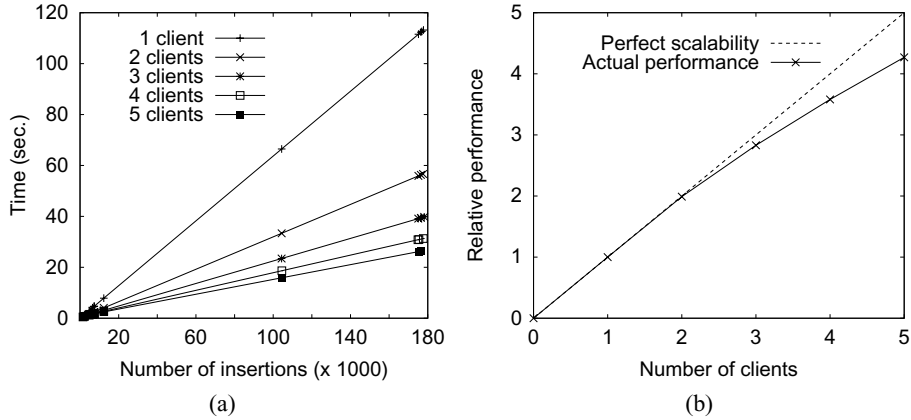


Fig. 2. Building the LH*LH SDDS: absolute performance (a) and scalability (b). In graph (a), the data points correspond to the moments in time where splits take place.

phenomenon is referred to as *cascading splits* [7]. It is caused by the fact that when the load factor on server n (the token-holder) is high enough to trigger a split, the servers that follow server n are often also ready to split. This means that after server n has split and forwarded the n token to the next server, the new token-holder immediately splits as well. As a result, a cascade of splitting servers is formed which terminates whenever a server is encountered with a load factor that is lower than the threshold. Essentially, cascading splits are undesirable as they harm the incremental fashion with which the distributed data structure is reorganised. Litwin et al. [7] have proposed several adjustments to the split threshold function to achieve a more uniform distribution of the splits.

Figure 3a plots the curves of the average time needed for a single insertion, as experienced by the application. The data points again refer to the moments in time where splits occur. Because the data points for the first 20,000 insertions are relatively hard

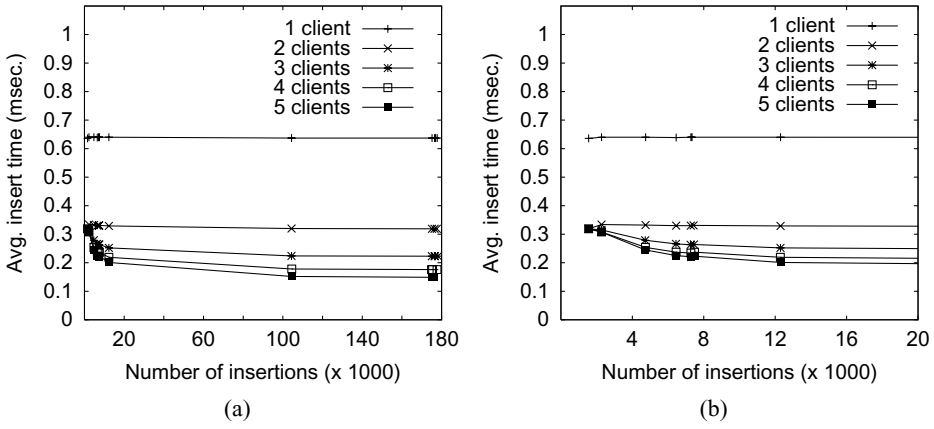


Fig. 3. Average time per insertion in milliseconds.

# Clients	# IAMs	Msg./insertion	Average message overhead
1	14	2.0008	0.04%
2	24	2.0011	0.06%
3	43	2.0012	0.06%
4	48	2.0014	0.07%
5	55	2.0014	0.07%

Table 1. Number of messages required to build the LH*LH SDDS.

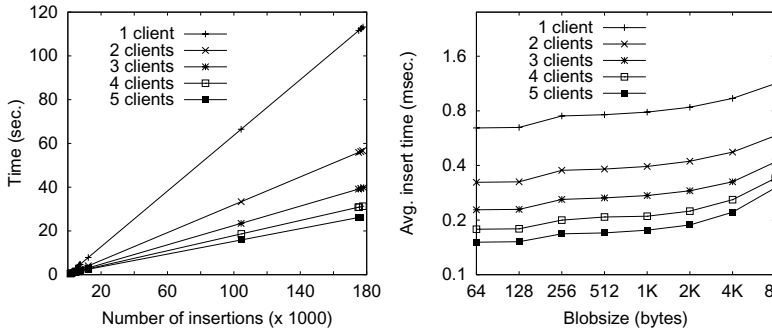
to distinguish, Figure 3b zooms in on this particular range. Figure 3a shows that the worst average insertion time (for 1 client) does not exceed 0.65ms. This is about one order of magnitude faster than the typical time to access a disk. However, we should remind the reader that these insertion latencies reflect a lower bound (for the investigated multicomputer architecture) since no computation is modelled at the clients and servers.

As can be expected from Figure 2a, Figure 3 shows that the average insertion time decreases when increasing the number of clients. Additionally, the average insertion time also decreases for an increasing number of insertions. This is especially true during the first few thousands of insertions and for the experiments using 3 or more clients. The reason for this is that a larger number of clients requires the creation of enough servers before the clients are able to effectively exploit parallelism, i.e. allowing the clients to concurrently access the LH*LH SDDS with low server contention. As can be seen in Figure 3b, after about 8,000 insertions, enough server processes have been created to support 5 clients.

In Table 1, the message statistics are shown for building the LH*LH SDDS. For about 180,000 insertions, the maximum number of IAMs (Image Adjustment Messages) that were sent is 55 (for 5 clients). This is only 0.003% with respect to the total number of insertions. The average number of messages per insertion is near the optimum of 2 (needed for the request itself and the acknowledgement). The last column shows the average proportional message overhead (due to IAMs and the forwarding of messages) for a single insertion. These results confirm the statement from Litwin et al. [7] in which they claim that it usually takes one message only to address the correct server.

In Figure 4, the results are shown when experimenting with the blobsize of data elements. Figure 4a plots the curves for the build-time when using a blobsize of 2Kb (the data points again correspond with the server splits). The results show that the build-times are still linear to the number of insertions and that the performance scales properly when adding more clients. In fact, the build-times are not much higher than the ones obtained in the experiment with empty data elements (see Figure 2a). The reason for this is twofold. First, since computational latencies like memory references are not simulated, our simulation model is especially optimistic for data elements with a large blobsize. Second, the modelled communication overhead of the AIX kernel, which is running on the real machine's nodes, is rather high (at least 325μs per message) and dominates the communication latency. This reduces the effect of the message size on the communication performance.

Figure 4b depicts the average time per insertion as experienced by the application when varying the blobsize from 64 bytes to 8Kb (the latter is building a data structure



(a) (b)

Fig. 4. Effect of the blobsize on the performance: the build-time when using a blobsize of 2Kb (a) and the effect of the blobsize on the average time per insertion (b).

of 1.4 Gbytes). Note that both axes have a logarithmic scale. Two observations can be made from this figure. First, the average insertion time suddenly increases after a blobsize of 128 bytes. This effect is somewhat diminished when using more clients. The reason for the performance anomaly is a peculiarity in the implementation of synchronous communication for the Parsytec CC machine. Messages smaller than 248 bytes can piggy-back on a fast initialisation packet, which sets up the communication between the source and destination nodes. Beyond these 248 bytes, normal data packets have to be created and transmitted, which slow down the communication (e.g. the software overhead is higher). By increasing the number of clients, the effect of the higher communication overhead of big messages ($>$ 248 bytes) is reduced due the parallelisation of the overhead.

A second observation that can be made from Figure 4b is that the curves are relatively flat for blobsizes below the 1Kb. Again, this can be explained by the fact that the large OS overhead dominates the insertion performance for small blobsizes. For insertions with large blobsizes ($>$ 1Kb), the network bandwidth and possibly the network contention become more dominant. To investigate whether or not the network contention plays an important role in the insertion performance, Figure 5 shows the average time that packets were stalled within the network. We should note that the variation of these averages is rather large, which implies that they should be interpreted with care. Nevertheless, the graph gives a good indication of the intensity of the network contention. From Figure 5 can be seen that the contention increases more or less linearly when increasing the blobsize in the case of 1 client. However, when using more clients, the contention starts to scale exponentially. The highest measured average block time equals to 31μs (8Kb blobsize with 5 clients), which is still $1/10^{th}$ of the 325μs software communication overhead for a single message. So, in our experiments, the contention is not high enough to dominate the insertion performance. This suggests that the insertion performance for large blobsizes (see Figure 4b) is dominated by the network bandwidth.

So far, we have assumed that the server (and client) processes are infinitely fast, i.e. computation is not modelled. To investigate the effect of server overhead on the overall performance, we modelled a delay for every incoming insertion on a server (the clients

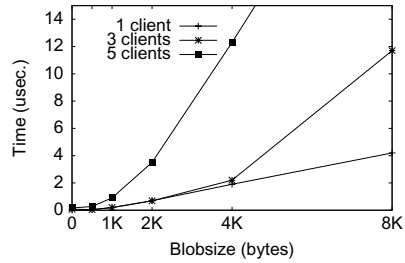


Fig. 5. The average time that packets blocked within the network.

continue to be infinitely fast). During the delay, which takes place after acknowledging the client's request, the server is inactive. Since the server overhead can overlap with the transmission of new client requests, this scheme exploits parallelism for all client configurations (even when 1 client is used). We varied the server overhead from 0 (no overhead) to 50ms per insertion.

Figure 6 depicts the results of the server overhead experiment. In Figure 6a, the effect on the average insertion time is shown. Note that both axes have a logarithmic scale. Figure 6a shows that the insertion latency starts to be seriously affected by the server overhead after a delay of approximately 0.1ms. Beyond a server overhead of 1ms, the insertion latency increases linearly with the server overhead which indicates that the insertion latency is entirely dominated by the server overhead. After this point, the differences between the various client configurations have more or less been disappeared as well, i.e. the curves converge. This implies that the large server overheads reduce the potential parallelism. An important reason for this is that when the server overhead approaches or exceeds the minimum time between two consecutive insertion requests from a single client (which is in our case simply the 325μs software communication overhead as the clients do not perform any computation), the rate at which a server can process insertion requests becomes lower than the rate at which a single client can produce requests. This means that a server can easily become a bottleneck when adding clients.

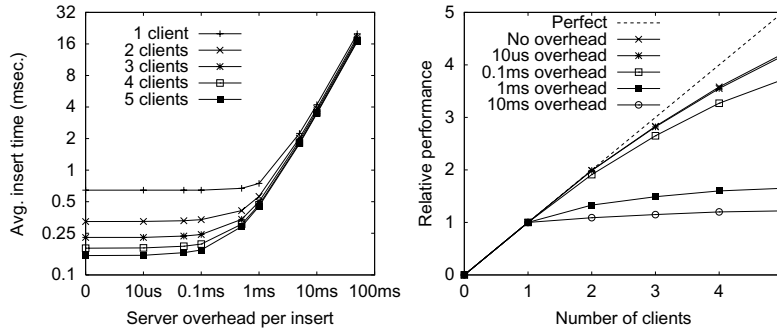


Fig. 6. Effect of the server overhead on the performance: the average time per insertion (a) and the scalability (b).

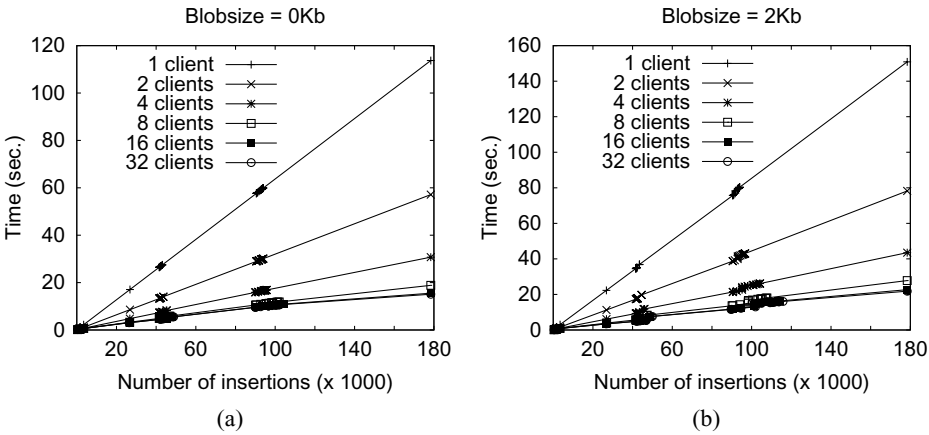


Fig. 7. Build-times for the 64-node Mesh of Clos network using blobsizes of 0 (a) and 2Kb (b).

The reduction of parallelism caused by large server overheads is best illustrated in Figure 6b. This figure shows the speedup for multiple clients when varying the server overhead. It is obvious that while the client-scalability is good for server overheads of 0.1ms and less, the scalability for larger overheads has collapsed completely.

Until now, the experiments have been performed using the simulation model of a 16-node Parsytec CC architecture. This has limited our simulations to allocating a maximum of 5 clients and 11 servers. To investigate how LH*LH behaves for a larger number of clients and servers, we have also simulated a 64-node Mesh of Clos network [10], which connects four multistage clusters of 16 nodes in a 2×2 mesh.

Figure 7 shows the build-times for this Mesh of Clos when using blobsizes of 0 (Figure 7a) and 2Kb (Figure 7b). Again, the data points refer to the moments in time where a split occurs. In these experiments, we have simulated the LH*LH for up to 32 clients. The curves in Figure 7 show the same behaviour as was observed in the experiments with the 16-node Parsytec CC model: the build-time is linear to the number of insertions and decreases with an increasing number of clients. The occurrence of cascading splits is also illustrated by the clusters of points in Figure 7.

In Figure 8, the scalability for several blobsizes is plotted by the curves which are labelled with *normal*. Additionally, the curves labelled with ”1/3” and ”1/9” show the scalability when the 325μs communication overhead is reduced by a factor 3 and 9 respectively. A number of observations can be made from Figure 8. First, the *normal* curves indicate that the configurations with blobsizes up to 2Kb scale to roughly 8 clients, whereas the configuration with a blobsize of 8Kb only scales up to about 4 clients. The deterioration of scalability for large blobsizes is due to the increased network contention.

Another, quite interesting, result is that the scalability for the large blobsizes is reduced even more when decreasing the software communication overhead. This effect is also caused by the increase of network contention: the smaller the overhead, the higher the frequency at which the clients inject insertion messages into the network.

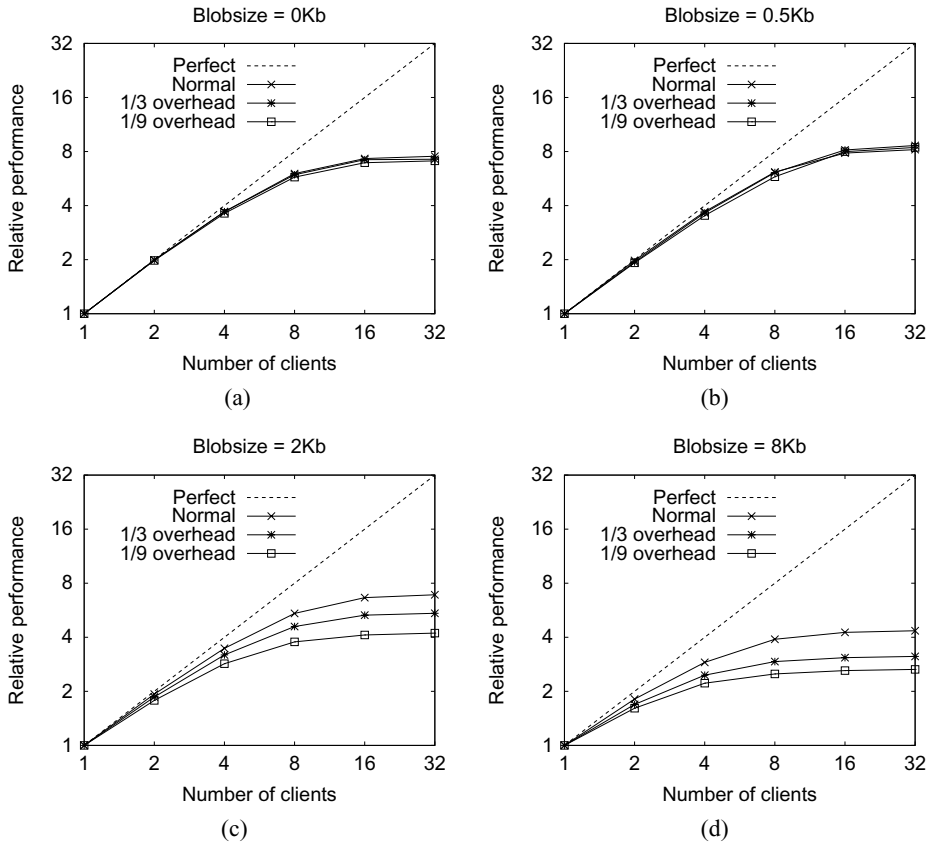


Fig. 8. The scalability for blobsizes of 0 bytes (a), 512 bytes (b), 2Kb (c) and 8Kb (d). The curve labelled with *normal* refers to the results of the original model. The other two curves (1/3 and 1/9) present the results for a modified model in which the software communication overhead is reduced by a factor of 3 and 9 respectively.

Clearly, the higher frequency of insertions with large blobsizes results in more network traffic and thus more contention. So, one can conclude that a part of the measured client-scalability for large blobsizes is due to the high communication overhead of the modelled platform. Apparently, the increase of network traffic in the case of a blobsize of 512 bytes (see Figure 8b) is not severe enough to cause a lot of extra contention.

6 Conclusions

In this study, we evaluated the LH*LH distributed data structure for a multicomputer architecture. Our simulation results show that LH*LH is indeed scalable: the time to insert a data element is independent on the size of the data structure. For the studied multicomputer, the insertion time as experienced by a single client can be an order of magnitude faster than a typical disk access. The insertion time can even be reduced by

using multiple clients which concurrently insert data into the distributed data structure. The results indicate that the speedup of insertions scales reasonably well up to about 8 clients for the investigated network architecture and workload.

We have also shown that the scalability of the insertion performance can be affected in more than one way. For instance, the larger the data elements that are inserted, the poorer is the scalability when increasing the number of clients. We found that the performance of insertions with data elements of 512 bytes scales up to 8 clients, whereas the performance of insertions with 8Kb data elements only scales up to 4 clients. Moreover, a large server overhead can seriously hamper the client-scalability of the LH*LH data structure. Our results indicate that it is important to keep the server overhead below the minimum time between two consecutive requests from a single client. This increases the potential parallelism as it ensures that the rate at which a server can process requests is higher than the rate at which a single client can produce requests. Finally, we found that a part of the speedup for multiple clients is due to the parallelisation of software communication overhead (which is quite large for the studied architecture). When the communication is optimised (i.e. the overhead is reduced), lower speedups are achieved.

Acknowledgements

We would like to thank Jonas Karlsson for his feedback regarding our LH*LH model.

References

[1] P. A. Boncz and M. L. Kersten. Monet: An impressionist sketch of an advanced database system. In *Proc. of IEEE BIWIT workshop*, July 1995.

[2] W. J. Dally and C. L. Seitz. The torus routing chip. *Journal of Distributed Computing*, 1(3):187–196, 1986.

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[4] J. S. Karlsson. A scalable data structure for a parallel data server. Master's thesis, Dept. of Comp. and Inf. Science, Linköping University, Feb. 1997.

[5] J. S. Karlsson, W. Litwin, and T. Risch. LH*lh: A scalable high performance data structure for switched m ulticomputers. In *Advances in Database Technology*, pages 573–591, March 1996.

[6] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of VLDB*, 1980.

[7] W. Litwin, M-A. Neimat, and D. Schneider. LH*: A scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–526, Dec. 1996.

[8] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26:62–76, Feb. 1993.

[9] A. D. Pimentel. *A Computer Architecture Workbench*. PhD thesis, Dept. of Computer Science, University of Amsterdam, Dec. 1998.

[10] A. D. Pimentel and L. O. Hertzberger. Evaluation of a Mesh of Clos wormhole network. In *Proc. of the 3rd Int. Conference on High Performance Computing*, pages 158–164. IEEE Computer Society Press, Dec. 1996.

[11] A. D. Pimentel and L. O. Hertzberger. An architecture workbench for multicomputers. In *Proc. of the 11th Int. Parallel Processing Symposium*, pages 94–99. IEEE Computer Society Press, April 1997.